CSA Frontend Framework Design

# Introduction

This framework design will focus on the layout of the CSA functionality integrated with Improve. CSA will be able to be accessed from Improve mainly as a tool to:

1. **Sales Analysis:** Generate sale analysis reports including CI sales analysis and vacant land sales analysis.
2. **Appraisal:** Create and view work sheet as well as do comp search.

Also this design will also lay ground to the security features within the CSA components. To make sure all pages will meet authorization requirements as well as data passing encryption to prevent unwanted direct access in the same provide a provide a solid test solution for the ease of testing.

# Core Technologies

Angular Framework 16.0

Angular Testing Utilities

Kendo Telerik UI

# Architectural Patterns

## Loading Strategy from Improve to CSA Components:

For this part there are less options as Improve code is not supposed to be changed. Thus, the data will be passed into CSA through query string. Security should be fine long as Improve make sure to check user credentials to prevent outside access. Extra precautions could be taken with the implementation of the Default page. Use default page to receive query string data passed by Improve, store the data and then re-route to destination page. Here we will use the default page as the parent page and destination page as child page. In this way data received in the default page will be easily passed down to its child page with the use of @Input decorator (detailed code example is included in below “From parent component to child component”). Through this implementation query string will only appear a split second in the loading page then stored and disappeared in the destination page.

## Data Passing Between Frontend Components:

Avoid using any query string to pass data due to security concerns. In the frontend Angular Framework should take advantage of the various data passing methods provided by Angular by default. Passing data via one of these three ways:

1. From parent component to child component: In the child component we could use **Input decorator.** Below is the sample code to pass in data from parent component and received by child component:

// child.component.ts

import { Component, Input } from '@angular/core';

@Component({

  selector: 'app-child',

  template: `

    <p>{{ receivedData }}</p>

  `,

})

export class ChildComponent {

  @Input() receivedData: string;

}

1. From child component to parent component: Use Output decorator and emit the data up to parent component. Below is the sample code of child component emitting data as well as parent component receiving data:

// child.component.ts

import { Component, EventEmitter, Output } from '@angular/core';

@Component({

  selector: 'app-child',

  template: `

    <button (click)="sendDataToParent()">Send Data to Parent</button>

  `,

})

export class ChildComponent {

  @Output() sendData: EventEmitter<string> = new EventEmitter<string>();

  sendDataToParent() {

    this.sendData.emit('Data from child');

  }

}

1. Use service for model data that are generally shared by multiple different components or in the case when two components do not share a parent-child relationship. This takes advantage of the dependency injection feature provided by angular. As data service created could be injected into component to use its members. Below is an example of an Angular data service:

// CSASalesInfo.service.ts

import { Injectable } from '@angular/core';

@Injectable({

  providedIn: 'root',

})

export class CSASalesInfoService {

  csaId: number;

  seqNum: number;

  commentText: string;

  entryTs: Date;

  entryWorker: string;

  updateTs: Date;

  updateWorker: string;

  setCSAData(data: any): void {

    this.csaId = data.CSA\_ID;

    this.seqNum = data.SEQ\_NUM;

    this.commentText = data.COMMENT\_TEXT;

    this.entryTs = data.ENTRY\_TS;

    this.entryWorker = data.ENTRY\_WORKER;

    this.updateTs = data.UPDATE\_TS;

    this.updateWorker = data.UPDATE\_WORKER;

  }

}

## Communication with APIs:

For communication between frontend and backend, use HTTP Client together with Observable to subscribe to data and reflect any changes if needed. Build service to specifically taken care of different HTTP requests following code gives a sample for consideration:

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root',

})

export class DataService {

  private apiUrl = 'https://api.example.com';

  constructor(private http: HttpClient) {}

  // GET request to fetch data

  getData(): Observable<any> {

    return this.http.get(`${this.apiUrl}/data`);

  }

  // POST request to add data

  addData(newData: any): Observable<any> {

    return this.http.post(`${this.apiUrl}/data`, newData);

  }

  // PUT request to update data

  updateData(id: number, updatedData: any): Observable<any> {

    return this.http.put(`${this.apiUrl}/data/${id}`, updatedData);

  }

  // DELETE request to delete data

  deleteData(id: number): Observable<any> {

    return this.http.delete(`${this.apiUrl}/data/${id}`);

  }

}

Inject the service to use these HTTP Client calls then another service should be created to hold incoming data. These services should be considered as DTOs, created in the folder DLL.

## Routing System:

Security Features: On default page if user cannot be logged in (cannot obtain a token). User will be route to the error page, as CSA should not perform any login for user instead user should be logged into Improve properly and route to usa page through correct access method (either through other functionality or using the side door with correct APN selected).

Default page will be used as a terminal to route user to specific CSA pages (Sale Analysis, Worksheet etc).

As the rest of the route CSA will use app-routing module. Below is an example of how to set it up:

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

import { HomeComponent } from './cisalesanalysis/cisalesanalysis.component';

import { AboutComponent } from './loginerror/loginerror.component';

const routes: Routes = [

  { path: 'CISalesAnalysis', component: CISalesAnalysisComponent },

  { path: 'LoginError', component: LoginErrorComponent },

];

@NgModule({

  imports: [RouterModule.forRoot(routes)],

  exports: [RouterModule],

})

export class AppRoutingModule {}

## Testing Concerns:

Query string is still available though only in “Default” loading page. We can still add query string after the /Default?. This should at least provide convenient access to all major pages such as SalesAnalysis and Appraisal pages. PBPage shall also been rebuilt, however it should be only available to programmers for debugging and testing purpost only

However, we could also consider testing a component using Angular Testing Utility. As it will be able to mimic injected data services to predict the component’s behavior and verify test result. Making this debugging process highly independent. Below is an example using testing utility to mimic an injected service’s data:

// data.component.spec.ts

import { ComponentFixture, TestBed } from '@angular/core/testing';

import { DataComponent } from './data.component';

import { DataService } from './data.service';

describe('DataComponent', () => {

  let component: DataComponent;

  let fixture: ComponentFixture<DataComponent>;

  beforeEach(() => {

    TestBed.configureTestingModule({

      declarations: [DataComponent],

      providers: [DataService],

    }).compileComponents();

  });

  beforeEach(() => {

    fixture = TestBed.createComponent(DataComponent);

    component = fixture.componentInstance;

  });

  it('should create', () => {

    expect(component).toBeTruthy();

  });

  it('should inject data from the service', () => {

    const dataService = TestBed.inject(DataService);

    spyOn(dataService, 'getData').and.returnValue('Mocked data');

    fixture.detectChanges();

    expect(component.data).toBe('Mocked data');

  });

});

Since for Angular, when generate a component using command such as “ng g c NewComponent” will automatically create a test doc for the existing component (the “spec.ts” file). Write tests will help fulfill any future test requirement as well as code coverage.

BLL as for business layer should only hold services that contain functionalities shared by different components. Each component will have its own TypeScript file to specify its functions.

# Configuration

Here since Angular does not have a built in Configuration file. We will set up environment file which performs essentially similar functions. For hardcoded information that will frequently change, these info should be stored in this environment file for the convenient of editing. As environment could be setup for development and production separately.

For development:

// src/environments/environment.ts

export const environment = {

  test: false,

  apiBaseUrl: 'https://improveapinewd.acgov.org',

// Only add this info if WindowsAuthentication is not enough

bearerToken: 'your-secret-bearer-token',

};

For production:

// src/environments/environment.test.ts

export const environment = {

  test: true,

  apiBaseUrl: 'https://improveapinewt.acgov.org',

// Only add this info if WindowsAuthentication is not enough

bearerToken: 'your-secret-bearer-token',

};

# Model Layer

In DAL folder, this folder should contain all necessary service component to store data that is passed between API and Web Application. Also within DAL folder there should be two separate folders: inDTO and outDTO. To differenciate models store update info that been send to api to make an database update or inDTO to store models that passed in from api to be displayed to client. As for detailed model contents, please refer to **entities and models** section of the api design above based which api is been called.

# Data Service

This will be the data service we use combine with API service below to make outbound calls communicating with backend apis.

// Import necessary modules

import { Injectable } from '@angular/core';

import { HttpClient, HttpHeaders } from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable({

  providedIn: 'root',

})

export class DataService {

  const headers = new HttpHeaders({ 'Content-Type': 'application/json' });

  const options = { withCredentials: true, headers: headers };

  constructor(private http: HttpClient) {}

  // Get method to fetch data from the API

  public getData(endpoint: string): Observable<any> {

    const url = `${endpoint}`;

    return this.http.get<any>(url, { withCredentials: true });

  }

  // Post method to send data to the API

  public postData(endpoint: string, data: any): Observable<any> {

    const url = `${endpoint}`;

    return this.http.post<any>(url, data, { withCredentials: true });

  }

  // Delete method to delete data from the API

  public deleteData(endpoint: string, id: number): Observable<any> {

    const url = `$${endpoint}/${id}`;

    return this.http.delete<any>(url, { withCredentials: true });

  }

  // Utility method to set headers with authorization token if needed

  // only use this if windows authentication alone is not enough, put

// this inplace of this.options

  private getHeaders(): { headers: HttpHeaders } {

    const authToken = 'your-auth-token'; // Replace with your authorization token if needed

    const headers = new HttpHeaders().set('Authorization', `Bearer ${environment.bearerToken}`);

    return { headers };

  }

}

# API Service

To make it easier for api calling, build an API service will help locating correct api rather than search all over places. Below is possible code for api service:

import { Injectable } from '@angular/core';

@Injectable({

  providedIn: 'root',

})

export class ApiService {

// Here the base URL can also be set use if block to check if this is

// test environment as above setup

  private baseUrl: string = environment.apiBaseUrl;

  private baseImproveApi = '${this.baseUrl}/api';

  private baseCSAApi = '${this.baseUrl}/csa/api'

  // Use Typescript property for better data encapsulation

  get loginApi(): string {

    return `${this.baseCSAApi}/LogIn`;

  }

get errorLoggingApi(): string {

    return `${this.baseCSAApi}/ErrorLogging`;

  }

}

# Exception Handling

For angular will follow similar pattern to log exceptions to MS sql server. Two error handling methods will be implemented.

## Error Message Store

All Angular error messages will be stored within Resource.json file. This file can be opened with text editor and make necessary changes, the message style will remain same as current Improve setup:

Text

Description automatically generated

To use these messages in Angular, CSA web app will setup translation service to handle these text. Code example below:

Translation service

import { Injectable } from '@angular/core';

import { TranslateService } from '@ngx-translate/core';

@Injectable({

  providedIn: 'root',

})

export class TranslationService {

  constructor(private translateService: TranslateService) {}

  getMessage(key: string, ...params: any[]): string {

    return this.translateService.instant(key, { ...params });

  }

}

In the component

import { Injectable } from '@angular/core';

import { TranslationService } from './translation.service';

@Injectable({

  providedIn: 'root',

})

export class YourComponentService {

  constructor(private translationService: TranslationService) {}

  someFunction() {

    const message = this.translationService.getMessage('exceeds', 'Field', '10');

    console.log(message); // Outputs: <li>Field exceeds length of 10.</li>

  }

}

## Global Error Handling with ErrorHandler

Angular provides the ErrorHandler service that you can extend to create a custom global error handler. This service catches errors throughout your application. Below would be how it can be setup:

import { Injectable, ErrorHandler } from '@angular/core';

@Injectable()

export class GlobalErrorHandler implements ErrorHandler {

  // inject below ErrorLoggingService to handle error

  constructor(private errorLoggingService: ErrorLoggingService) {}

  handleError(error: any): void {

    // Handle the error with logError function in injected service, assume 10

// 10 will be used for angular exception logging and 2 is error code

    errorLoggingService.logError(10, 2, “generic exception”);

  }

}

Don’t forget to register it in AppModule

providers: [

  { provide: ErrorHandler, useClass: GlobalErrorHandler }

]

Then we will use try catch block to catch exceptions and log more detailed error messages. While error messages should be send to Improve api to store in MS sql server. Below is what the service for error logging might look like:

import { Injectable } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { Observable, throwError } from 'rxjs';

import { catchError } from 'rxjs/operators';

@Injectable({

  providedIn: 'root',

})

export class ErrorLoggingService {

// This service is designed to be used with apiSerice

  constructor(private http: HttpClient) {}

  logError(apiUrl: string, applicationId: number, logTypeId: number, customMessage: string): Observable<any> {

    const errorData = {

      applicationId: applicationId,

      logTypeId: logTypeId,

      customMessage: customMessage

    };

    return this.http.post(apiUrl, errorData).pipe(

      catchError((error) => {

        console.error('Error Logging API request failed:', error);

        return throwError(error);

      })

    );

  }

}

## Within Component use Try-Catch Block

Use try catch block when is required especially when doing IO operations. Throw the error and use the above error logging service to log the error to corresponding database. Blow is the example for try-catch block setup

import { Component, OnInit } from '@angular/core';

import { ErrorLoggingService } from './errorlogging.service';

@Component({

  selector: 'app-my-component',

})

export class MyComponent implements OnInit {

  constructor(private errorLoggingService: ErrorLoggingService,

    private apiService: ApiService) {}

  ngOnInit() {

    try {

      // Code that might throw an error

      this.performSomeOperation();

    } catch (error) {

      // Handle the error

      this.errorLoggingService.logError(apiService.errorLogging, 10, 1, "generic error")

    }

  }

  performSomeOperation() {

    // Operation details

  }

}

## Error Code Handling in Angular

Use Http Interceptor of Angular to interceptor error code

// auth-interceptor.ts

import { Injectable } from '@angular/core';

import {

  HttpRequest,

  HttpHandler,

  HttpEvent,

  HttpInterceptor,

  HttpResponse,

  HttpErrorResponse

} from '@angular/common/http';

import { Observable, throwError } from 'rxjs';

import { catchError } from 'rxjs/operators';

@Injectable()

export class AuthInterceptor implements HttpInterceptor {

  constructor() {}

  intercept(request: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

    return next.handle(request).pipe(

      catchError((error: HttpErrorResponse) => {

        if (error.status === 401) {

          // Handle 401 error (redirect to error page)

        }

        return throwError(error);

      })

    );

  }

}

Register the interceptor

// app.module.ts

import { NgModule } from '@angular/core';

import { BrowserModule } from '@angular/platform-browser';

import { HttpClientModule, HTTP\_INTERCEPTORS } from '@angular/common/http';

import { AppComponent } from './app.component';

import { AuthInterceptor } from './auth-interceptor'; // Import your interceptor

@NgModule({

  declarations: [AppComponent],

  imports: [BrowserModule, HttpClientModule],

  providers: [

    // Register the interceptor

    { provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptor, multi: true }

  ],

  bootstrap: [AppComponent]

})

export class AppModule {}

# Security

For CSA only authenticated user should have access to its feature.

For the backend as we are implementing classes within Improve api, which is using JWT token to secure its endpoints, so CSA server side can simply take advantage of this existing structure.

In Angular CSA will be using a token based security system. Token will be distributed from default page. Once default page is entered a token will be provided to current user and stored in their local storage with a given expiration time (1 hour as a default). Other CSA pages will have to check token in order to be accessed, otherwise a 401 error should be thrown.

Auth service is already implemented in CSA Web App under the Services folder. As well as set token function within the default page. For all other CSA page (except PBPage) will require to inject auth service and check isAuthenticated function return.

If backend return 401 Unauthorized back, in Angular it will be handled by HttpInterceptor. Check Exception Handling => Error Code Handling in Angular for more detail.

# Dynamically Changed Components:

Shared components such as navigation bar should not be hardcoded. Instead, it should be able to switch its content based on the user type currently logged in. Build a service to store user-type information. Another service should also be created in order to connect to backend MenuAPI. Here MenuAPI should query IE\_CSA\_MENU database to obtain menu option and return menu items in order to achieve better loose coupling between components.

Menu service should be injected to generate the tool bar component and inject the user service. Use lifecycle hook “ngOnInit” to set up. Then in the context page loop through the menu items received from MenuAPI for a dynamic items display. Below are the component file and context file example:

*Component file (ts file)*

// menu-bar.component.ts

import { Component, OnInit } from '@angular/core';

import { MenuService } from './menu.service';

@Component({

  selector: 'app-menu-bar',

  templateUrl: './menu-bar.component.html',

})

export class MenuBarComponent implements OnInit {

  menuItems: MenuItem[];

  constructor(private menuService: MenuService) {}

  ngOnInit(): void {

    this.menuItems = this.menuService.getMenuItems();

  }

}

interface MenuItem {

  id: number;

  label: string;

  route: string;

}

*Context file (html file)*

<!-- menu-bar.component.html -->

<div>

  <app-menu-item \*ngFor="let menuItem of menuItems" [menuItem]="menuItem"></app-menu-item>

</div>

Also consider modifying styling(css) file to keep a consistent design from Improve webpages.

# Menu Design (IE\_CSA\_MENU schema)

## Navigation Bar Design (Kendo) For all component

For CSA Sales Analysis pages (C/I Improved Property Analysis and Vacant Land Analysis pages). Since both pages are specifically used for doing analysis with selected property, thus there won’t be any need of a navigation menu.

For Appraisal page similar data table structure will be implemented as Improve and CTM. Here will use parent child relation menu. As there will be two layers, parent menu item will be using 0 as its MENU\_ID\_LEVEL1 row entry. Nested menu options (child menu items) will be set to 1.

Below is the menu item data table design:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| MENU\_ID | MENU\_ID\_LEVEL0 | LEVEL1 | LEVEL2 | MENU\_NAME | MENU\_URL | ENABLED\_FL | APN\_FL |
| 1 | 1 | 0 | 0 | Cost Approach | /wkSheetCost | Y | Y |
| 2 | 2 | 0 | 0 | Vacant Land Comps |  | Y | Y |
| 3 | 3 | 0 | 0 | Improved Property Comps |  | Y | Y |
| 4 | 4 | 0 | 0 | Income Approach |  | Y | Y |
| 5 | 5 | 0 | 0 | Reports | /csaReports | Y | Y |
| 6 | 6 | 0 | 0 | Exit |  | Y | Y |
| 7 | 2 | 1 | 1 | New Worksheet | /wkSheetNewImps | Y | Y |
| 8 | 3 | 1 | 1 | New Worksheet | /wkSheetNewImps | Y | Y |
| 9 | 4 | 1 | 1 | New Worksheet | /wkSheetNewImps | Y | Y |
| 10 | 2 | 1 | 2 | Existing Worksheet | /wkSheetList | Y | Y |
| 11 | 3 | 1 | 2 | Existing Worksheet | /wkSheetList | Y | Y |
| 12 | 4 | 1 | 2 | Existing Worksheet | /wkSheetList | Y | Y |
| 13 | 2 | 1 | 3 | Comp Search | /VLcompSearch | Y | N |
| 14 | 3 | 1 | 3 | Comp Search | /VLcompSearch | Y | N |
| 15 | 4 | 1 | 3 | Comp Search | /VLcompSearch | Y | N |

Then for Angular, we will use Kendo-Menu to display all this nav bar items. Below is the code example of how to import menu items from data base and display them. Remember to edit menu style to make it fit the design of the Improve:

import { Component, OnInit } from '@angular/core';

import { HttpClient } from '@angular/common/http';

import { DataService } from './data.service';

import { ApiService } from './api.service';

@Component({

  selector: 'app-menu',

  template: `

    <kendo-menu [items]="menuItems"></kendo-menu>

  `,

})

export class MenuComponent implements OnInit {

isDisabled: boolean = false;

  menuItems: any[] = [];

  constructor(private dataService: DataService,

    private apiService: ApiService) {}

  ngOnInit(): void {

    this.dataService.getMenuItems(this.apiService.GetMenuItemsApi).subscribe((items) => {

Also check if return items is empty. Then set this.isDisabled to true

      Before assign to this.menuItems, make sure to check each menu item returned its ENABLED\_FL is set to true based on above menu table design

    });

  }

}

Finally, add this component to all CSA pages, if it’s currently not in use (does not have a menu set up for it) based on above code in ngOnInit. The nav bar component should be automatically disabled. Menu item should only be displayed when its ENABLED\_FL is set to true.

If bool value is false, Sale Analysis component editable contents should be disabled to cooperate with this input parameter.

As a summary following steps should be followed when routing to make sure page can be dynamically changed:

**Service Created to Save Init Parameters ->**

**Component injected service and ngOnInit to assign value to local variable ->**

**Context display page based on its component’s local variable value**

## Tool Bar (Shared Component) Design

Keep Tool Bar design simple and use as a child component to any tab strip page. Detailed Save functions for different tabs in the Kendo Tab-Strip should be included in pages such as SalesAnalysisComponent that includes all tabs.

For tool bar only thing it should perform as a child component is emit EventEmitter as @Output depend on which button is clicked. While SaleAnalysis page should perform corresponding functions based on the EventEmitter coming from controller-bar child component as well as selectedTab in Kendo Tab Strip.

TypeScript:

import { Component, EventEmitter, Output } from '@angular/core';

@Component({

  selector: 'app-controller-bar',

  templateUrl: './controller-bar.component.html',

  styleUrls: ['./controller-bar.component.css']

})

export class ControllerBarComponent {

  @Output() saveClicked = new EventEmitter<void>();

  @Output() refreshClicked = new EventEmitter<void>();

  // Method to emit save event

  public onSaveClick(): void {

    this.saveClicked.emit();

  }

  // Method to emit refresh event

  public onRefreshClick(): void {

    this.refreshClicked.emit();

  }

}

HTML:

<div class="controller-bar">

  <button (click)="onSaveClick()">Save</button>

  <button (click)="onRefreshClick()">Refresh</button>

</div>

Sample for component that include kendo tab strip (SalesAnalysisComponent HTML):

<kendo-tabstrip [selectedTab]="selected">

  <kendo-tabstrip-tab [title]="'Tab 1'">

    Content of Tab 1

  </kendo-tabstrip-tab>

  <kendo-tabstrip-tab [title]="'Tab 2'">

    Content of Tab 2

  </kendo-tabstrip-tab>

  <kendo-tabstrip-tab [title]="'Tab 3'">

    Content of Tab 3

  </kendo-tabstrip-tab>

</kendo-tabstrip>

<app-controller-bar

  (saveClicked)="saveData()"

  (refreshClicked)="refreshData()">

</app-controller-bar>

# Kendo Telerik Integration

For this application Kendo Telerik will be integrated for UI support in order to keep a consistent design with Improve (with Telerik). We will be mainly using Kendo-Tabstrip for current tab control and Kendo-Grid for form control.

## Tabs

To keep changes in tab is stored in its parent component tab manager component, refer to above component toolbar design for details

Template:

<!-- cisalesanalysis.component.html -->

<kendo-tabstrip [selectedTab]="selected" (tabSelect)="onSelect($event)">

  <kendo-tabstrip-tab title="Sale Info">

    <ng-template kendoTabContent>

      <!-- Content for Sale Info tab -->

      <p>Content for Sale Info</p>

    </ng-template>

  </kendo-tabstrip-tab>

  <kendo-tabstrip-tab title=" Prop Chraracteristics">

    <ng-template kendoTabContent>

      <!-- Content for Prop Char tab -->

      <p>Content for Prop Char</p>

    </ng-template>

  </kendo-tabstrip-tab>

</kendo-tabstrip>

TypeScript:

// cisalesanalysis.component.ts

import { Component, EventEmitter, Output } from '@angular/core';

@Component({

  selector: 'app- cisalesanalysis',

  templateUrl: './ cisalesanalysis.component.html',

  styleUrls: ['./ cisalesanalysis.component.css'],

})

export class TabstripComponent {

  @Output() tabSelected: EventEmitter<number> = new EventEmitter<number>();

  public selected = 0;

  public onSelect(e) {

    this.selected = e.index;

    this.tabSelected.emit(this.selected);

  }

}

## Grid

<!-- Import necessary modules in your component -->

<kendo-grid [data]="gridData" [height]="400">

  <kendo-grid-column field="productName" title="Product Name"></kendo-grid-column>

  <kendo-grid-column field="category" title="Category"></kendo-grid-column>

  <kendo-grid-column field="unitPrice" title="Unit Price"></kendo-grid-column>

  <kendo-grid-column title="Actions" width="150">

    <ng-template kendoGridCellTemplate let-dataItem>

      <button kendoButton icon="edit" class="edit-button" (click)="editItem(dataItem)"></button>

      <button kendoButton icon="delete" class="delete-button" (click)="deleteItem(dataItem)"></button>

    </ng-template>

  </kendo-grid-column>

</kendo-grid>

In this TypeScript file

import { Component, OnInit } from '@angular/core';

import { DataService } from './data.service';

import { ApiService } from './api.service';

@Component({

  selector: 'app-example',

  templateUrl: './example.component.html',

  styleUrls: ['./example.component.css'],

})

export class ExampleComponent implements OnInit {

  public gridData: any[] = [];

  constructor(private dataService: DataService,

    private apiService: ApiService) {}

  ngOnInit(): void {

    // Fetch data from your API

    this.gridData = this.dataService.GetData(this.apiService.getLeaseApi)

  }

  public editItem(dataItem: any): void {

    // Implement edit logic here

    console.log('Edit item:', dataItem);

  }

  public deleteItem(dataItem: any): void {

    // Implement delete logic here

    console.log('Delete item:', dataItem);

  }

}

# Form Control

For form with input field this app will use Angular Reactive form’s FormBuilder. As an easy way to create a form as well as add necessary validation. Also can easily prevent the use of two way binding of template form to match client requirements.

In each tab, this app should be using Angular FormBuilder together with a separate folder for validator services. Note that CSA won’t validate each value as type but instead will do a validation for the whole tab input after save() function is called. Corresponding validation service will be called and perform a whole tab validation. ALL error message will be show together in a pop up message box. If there’s not validation issue, api call will be performed to do data modification.

Validator service with custom validators example:

import { Injectable } from '@angular/core';

import { AbstractControl, ValidatorFn, Validators } from '@angular/forms';

@Injectable({

  providedIn: 'root',

})

export class ValidatorService {

  constructor() {}

  validateMaxLength(maxLength: number): ValidatorFn {

    return (control: AbstractControl): { [key: string]: any } | null => {

      if (Validators.required(control)) {

        // If the control is required and empty, don't perform maxLength validation

        return null;

      }

      const value: string = control.value;

      if (value && value.length > maxLength) {

        return { maxLength: { requiredLength: maxLength,

          actualLength: value.length,

          message: 'Max length: ' + maxLength + 'exceded' } };

      }

      return null; // Validation passed

    };

  }

  validateExactLength(exactLength: number): ValidatorFn {

    return (control: AbstractControl): { [key: string]: any } | null => {

      if (Validators.required(control)) {

        return null;

      }

      const value: string = control.value;

      if (value && value.length != exactLength) {

        return { exactLength: { requiredLength: exactLength,

          actualLength: value.length,

          message: 'Need to be ' + exactLength + 'characters'  } };

      }

      return null;

    };

  }

  validateCustomEmail(): ValidatorFn {

    return (control: AbstractControl): { [key: string]: any } | null => {

      if (Validators.required(control)) {

        return null;

      }

      // Regular expression for a simple email validation

      const emailRegex = /^[a-zA-Z0-9.\_-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,4}$/;

      if (!emailRegex.test(control.value)) {

        return {

          invalidEmail: {

            valid: false,

            message: 'The provided email address is not in a valid format.'

          },

        };

      }

      return null;

    };

  }

  validateForm(form: { [key: string]: AbstractControl }): ValidationErrors[] {

    const validationErrors: string[] = [];

    Object.keys(form).forEach((controlName: string) => {

      const control: AbstractControl = form[controlName];

      if (control && control.invalid && (control.dirty || control.touched)) {

        validationErrors.push(control.errors[controlName].message)

      }

    });

    return validationErrors;

  }

}

Form builder example:

In Context:

<form [formGroup]="myForm">

  <label for="email">Email:</label>

  <input id="email" formControlName="email" />

  <label for="phone">Phone:</label>

  <input id="phone" type="phone" formControlName="phone" />

  <button type="submit" [disabled]="myForm.invalid">Submit</button>

</form>

In TypeScript:

import { Component, OnInit } from '@angular/core';

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

// Below is custom build email validator showed in above example

import { emailValidator } from './validators/email-validator';

// Data service using HTTP Client to retrieve data

import { DataService } from './data.service';

// API service will plainly used to provide api url to be used by data service

import { ApiService } from './api.service';

import { ErrorLoggingService } from './errorlogging.service';

// For details about api service, data service and error logging service

// please refer to previous section of the design doc

// This is the model used to store outbound DTO

import { CSASourceModel } from './DTO/out/CSASourceModel.service';

@Component({

  selector: 'app-csa',

})

export class YourComponent implements OnInit {

  myForm: FormGroup;

  constructor(private fb: FormBuilder,

private csaSourceModel: CSASourceModel,

private validatorService: ValidatorService,

private dataService: DataService,

    private apiService: ApiService,

    private errorLoggingService: ErrorLoggingService) {}

  ngOnInit(): void {

    // Initialize the form model with validators

    this.exampleForm = this.fb.group({

      SRCE\_EMAIL\_SEC: ['', [Validators.required, validatorService.validateExactLength(10)]],

      SOURCE\_PHONE\_SEC: ['', [Validators.required, validatorService.emailValidator()]],

    });

    // Fetch data from the service and populate the form

    // the keys in data should match the form control names

    this.dataService.getData().subscribe((data) => {

      this.myForm.patchValue(data);

    });

  }

  onSubmit(): void {

this.validatorService.validateForm(this.exampleForm.controls);

    if (this.myForm.valid) {

      // Use data service with api service to call backend api to update db

    } else {

      // Use error logging and api services to log this issue

    }

  }

  // Convenience getters for easier access to form controls in the template

  get email() {

    return this.exampleForm.get('email');

  }

  get phone() {

    return this.exampleForm.get('phone');

  }

}

For all the validation errors, display them as a pop up to list them all out. To achieve this we need a pop up window component as well as a service to open it, code example is below:

For pop up window component

Template:

<h1>Please correct errors in below items</h1>

<div \*ngFor="let error of validationErrors">

  {{ error }}<br>

</div>

<button mat-raised-button color="primary" (click)="close()">Close</button>

TypeScript

import { Component, Inject } from '@angular/core';

import { MAT\_DIALOG\_DATA, MatDialogRef } from '@angular/material/dialog';

@Component({

  selector: 'app-validation-errors-modal',

})

export class ValidationErrorsModalComponent {

  constructor(

    public dialogRef: MatDialogRef<ValidationErrorsModalComponent>,

    @Inject(MAT\_DIALOG\_DATA) public validationErrors: any[]

  ) {}

  close(): void {

    this.dialogRef.close();

  }

}

For service to open up the pop up window component

import { Injectable } from '@angular/core';

import { MatDialog } from '@angular/material/dialog';

import { ValidationErrorsModalComponent } from './validation-errors-modal.component';

@Injectable({

  providedIn: 'root',

})

export class ValidationErrorsModalService {

  constructor(private dialog: MatDialog) {}

  open(validationErrors: any[]): void {

    this.dialog.open(ValidationErrorsModalComponent, {

      data: { validationErrors },

      width: '400px',

    });

  }

}

# Report Handling

## Report creation

Report will be created with get methods specified in design doc data sources, use Visual Studio to design the new report with provided data.

## Report data input

Call backend api to get and update report based on current information from each tab.

## Report data display

After necessary data is injected into the report, locate PopReportViewerComponent in Angular app and set postParams to pinpoint the wanted report to display, and run getReport to get report to display in the pop up page. To clarify, the pop up page will simply display iframe that shows report display page (an ASPX) page from the Power BI report server, Angular is only used to command server to fetch the correct report, and display the snapshot using Power BI server webpage.